Physically-based Environment and Area Lighting using Progressive Rendering in WebGL

Philipp Otto
philipp.otto2@hpi.de
Hasso Plattner Institute,
Faculty of Digital Engineering,
University of Potsdam, Germany

Daniel Limberger
daniel.limberger@hpi.de
Hasso Plattner Institute,
Faculty of Digital Engineering,
University of Potsdam, Germany

Jürgen Döllner
juergen.doellner@hpi.de
Hasso Plattner Institute,
Faculty of Digital Engineering,
University of Potsdam, Germany

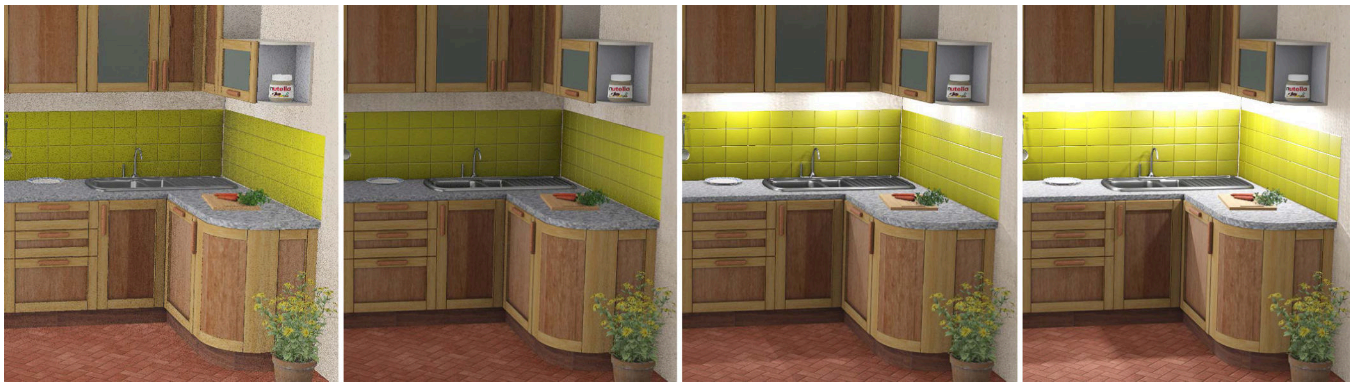


Figure 1: Intermediate frames of the kitchen model using our progressive renderer with a total frame count of 256 (right).

ABSTRACT

This paper presents a progressive rendering approach that enables rendering of static 3D scenes, lit by physically-based environment and area lights. Multi-frame sampling strategies are used to approximate elaborate lighting that is refined while showing intermediate results to the user. The presented approach enables interactive yet high-quality rendering in the web and runs on a wide range of devices including low-performance hardware such as mobile devices. An open-source implementation of the described techniques using TypeScript and WebGL 2.0 is presented and provided. For evaluation, we compare our rendering results to both a path tracer and a physically-based rasterizer. Our findings show that the approach approximates the lighting and shadowing of the path-traced reference well while being faster than the compared rasterizer.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; • **Human-centered computing** → *Ubiquitous and mobile computing*.

KEYWORDS

WebGL, progressive rendering, physically-based lighting, area lights, screen-space specular occlusion, real-time, image-based lighting

ACM Reference Format:
Philipp Otto, Daniel Limberger, and Jürgen Döllner. 2020. Physically-based Environment and Area Lighting using Progressive Rendering in WebGL. In *The 25th International Conference on 3D Web Technology (Web3D '20), November 9–13, 2020, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3424616.3424697

1 INTRODUCTION

Using physically motivated rendering concepts is an important part of photo-realistic rendering. These concepts incorporate physically-based reflection models, indirect lighting as well as the use of physical units for handling light intensity. Due to the continuous increase in performance of rendering hardware, these concepts are considered for real-time rendering more and more. We are now at a point, where mobile hardware is capable of running some of these physically motivated techniques. In offline rendering, these concepts have been used for a long time [Colbert et al. 2010]. In interactive real-time computer graphics applications, however, such techniques are prohibitively expensive. As a result, approximations are used extensively in order to attain physically plausible results [Karis and Games 2013; Lagarde and De Rousiers 2014].

With a broad support for OpenGL ES 3.0 and WebGL 2.0 [Jackson and Gilbert 2015], physically-based rendering has become feasible in the web and on mobile devices. In comparison to desktop rendering, the support for physically-based concepts by many web rendering frameworks is still limited though. The low performance

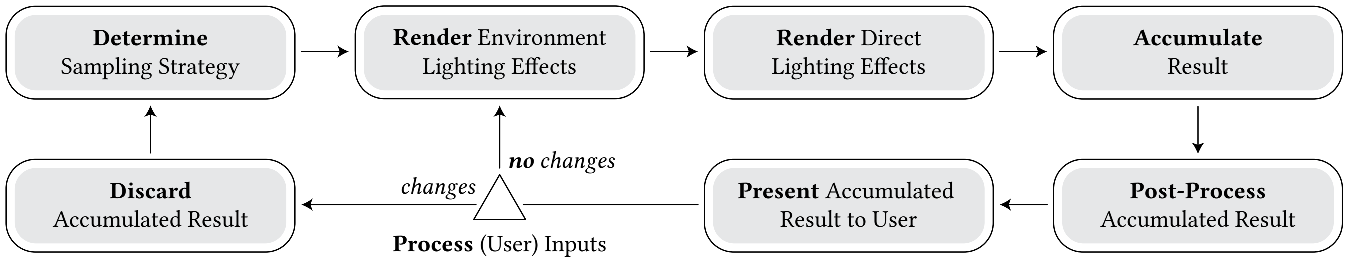

Determine Sampling Strategy
Render Environment Lighting Effects
Render Direct Lighting Effects
Accumulate Result
Post-Process Accumulated Result
Present Accumulated Result to User
***no** changes*
changes
Process (User) Inputs
Discard Accumulated Result

Figure 2: Workflow for progressive, physically-based rendering supporting environment and area lighting.

of mobile graphics cards render most of the real-time approximations of today's game engines too expensive for use in the web. In this work, we use progressive rendering [Limberger et al. 2016] for interactive, yet physically-based rendering in the web. Progressive rendering distributes the rendering workload across multiple frames while showing intermediate results to the user. It enables interactive and high quality rendering at the cost of temporal coherency and is mainly useful for static 3D geometry with no moving objects or lights. We show how (1) area lights based on Karis' [Karis and Games 2013] approach as well as (2) lighting by pre-filtered diffuse and specular cubemaps can be used in a progressive rendering setting. In addition, we present a rendering technique for progressive computation of both diffuse and specular occlusion based on this pre-filtered environment lighting. Besides our open source implementation, we provide a tool and discuss a tweak for pre-computation and storage of environment maps for image based lighting and web-based provisioning respectively. Finally, we discuss strategies on workload distribution that allow smooth interaction on weaker hardware such as mobile devices.

A renderer capable of producing interactive, high-quality renderings of static scenes can be used for a number of web-based applications such as product configurators, online asset marketplaces or online modelling toolkits. Additionally, a progressive renderer can be used to generate data sets for machine learning purposes. This can be achieved by rendering the same scene with a high number of samples and a low number of samples. The resulting images can then be used to train a network that is able to transform noisy images rendered at low sample counts to higher quality in real time. Such an approach is used by [Chaitanya et al. 2017] in order to reconstruct noisy images generated using Monte Carlo methods.

The goal of this work is to match the appearance of offline rendering results as closely as possible on mobile devices, while maintaining interactivity, e.g., for navigating the scene. Therefore, we compare our results to both a rasterizer and a path tracer in regards to visual quality and rendering time.

2 RELATED WORK

For related work, we discuss techniques of two areas, progressive rendering and physically-based rendering.

Progressive rendering. Distributing samples of a rendering technique across multiple frames to enhance interactivity was first used by [Fuchs et al. 1985] to facilitate anti-aliasing. This concept of *progressive rendering* was extended by [Haeberli and Akeley 1990], who introduced the *accumulation buffer* in order to render effects such as anti-aliasing, soft shadows, and motion blur by rendering the same 3D geometry multiple times and accumulating the results of all intermediate frames. In comparison to rendering the effects within one frame, intermediate results can be shown to the user, which allows smoother interaction with the scene. [Limberger et al. 2016] took this idea one step further and showed how to implement order-independent transparency, depth of field, screen-space ambient occlusion, soft-shadows and more. This was later used to produce high-quality renderings of geo-referenced data [Limberger et al. 2017]. [Sunet and Vazquez 2016] discuss approaches to optimize calculation of screen space ambient occlusion for mobile devices. They suggest using progressive rendering in order to improve performance. Multiple works on anti-aliasing are using temporal reprojection to use the results of the last frame for enhanced reconstruction, e.g., SMAA [Jiménez et al. 2012] and the Unreal Engine anti-aliasing implementation [Karis 2014]. Temporal reprojection has been shown to work very well for low-frequency effects such as ambient occlusion and indirect lighting. We decide against using it for our approach because for high-frequency direct lighting, the usual "ghosting" artifacts would be very noticeable. Also it could lead to "half-lit" scenes during navigation when some parts of the image were successfully reprojected while others were not.

Physically-based rendering. The microfacet model [Cook and Torrance 1982] was introduced as a way to model light reflection in a physically-based way. [Walter et al. 2007] built on this concept and introduced a new distribution term, the GGX distribution. For specular reflections, we use the Cook-Torrance model with the GGX distribution within our work. Based on the microfacet mordel, [Burley and Studios 2012] designed a BRDF model that uses the parameters *roughness, metallicness* and *base color* to render a broad range of materials. [Blinn and Newell 1976] introduced a way of generating reflections of environment lighting on curved surfaces, based on texture mapping. They store environment lighting data in textures and assume surfaces as perfect mirrors to obtain reflections. [Greene 1986] introduced the concept of the *Cube Map* as a more efficient representation of environment light in textures. Additionally, they describe how to filter cube maps based on the material properties and how to approximate this filtering process by pre-filtering the cube map via mip mapping. This concept is used in our work to approximate environment lighting on surfaces with different material properties. [Scheuermann and Isidoro 2006]

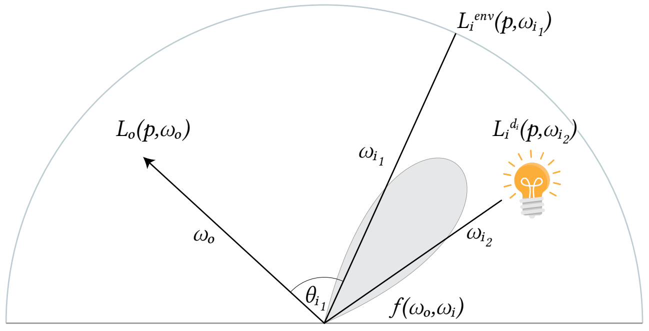


Figure 3: Illustration of outgoing radiance $L_o(p,\omega_o)$, incoming radiance from a distant environment $L_i^{\mathbf{env}}(p,\omega_{i_1})$ and incoming direct lighting radiance $L_i^{d_i}(p,\omega_{i_2})$. Notice that ω_{i_1} and ω_{i_2} are only single solid angles out of the integral over the hemisphere Ω.

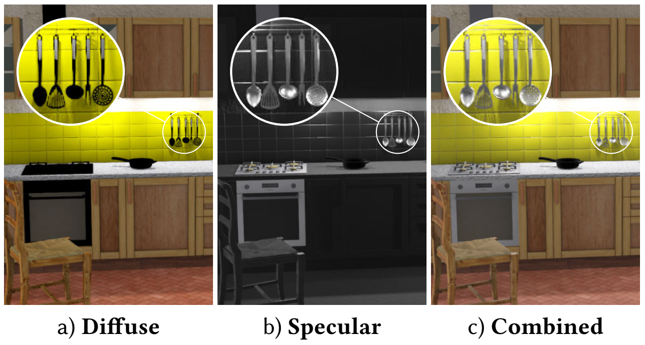


Figure 4: Visualization of the diffuse and specular lighting components of a rendering of a kitchen.

introduced the tool *CubeMapGen*, which implements a pre-filtering of cubemaps for different BRDFs as described by Greene. We implement a comparable tool (open source) that runs in the web to perform pre-processing of environment textures[1]. Karis [Karis and Games 2013] introduce the *split sum approximation* to approximate environment lighting using the GGX BRDF in a physically plausible way using only on environment map sample, which we use for the purpose of specular environment lighting. Additionally, he shows an approach to approximate area lights using the GGX BRDF by considering a area lights as a point lights that are located as close to the reflection vector as possible. With *screen space ambient occlusion* (SSAO) [Mittring 2007] an approach to render ambient occlusion as a post-processing effect was introduced. *Screen-space directional occlusion* [Ritschel et al. 2009] improves on SSAO by considering the direction of occlusion and thus weights the environment lighting in a more physically correct way. Additionally, one bounce of light is implemented by sampling the previous frame of the scene. By generalizing screen-space occlusion for specular BRDFs, [Jiménez et al. 2016] describe a way to render specular occlusion in real-time. We will show a technique to adapt these screen-space occlusion approaches to progressive rendering. [Sturm et al. 2016] introduced extensions to glTF and X3D to support physically-based rendering using those formats. These extensions have made their way into the glTF [Robinet and Cozzi 2013] standard, which we use for transmission and loading of 3D assets. Applications of physically-based rendering in the web have been published as part of ThreeJS [2], as well as VirtualLightJS [3] by Guthmann François. However, these demos show rather simple lighting scenarios and the shadowing is not physically modelled. Also for VirtualLightJS aliasing artifacts are quite visible, which we want to overcome with our approach.

3 TECHNIQUE

For our progressive rendering approach we propose a workflow that, prior to the synthesis of intermediate frames, determines a sampling strategy for rendering lighting effects. For each intermediate frame, environment lighting effects and direct lighting effects are then evaluated independently from one another. The results of these calculations are then accumulated with the results of previous intermediate frames, a common operation in progressive renderers. After a post-processing step is applied, the result is presented to the user, before continuing with the next intermediate frame and starting this process again. This process is depicted in Figure 2.

To calculate the radiance emitted from a point p within a 3D scene, Pharr et al. [Pharr et al. 2016] give:

$$L_o(p,\omega_o) = \int_\Omega f(\omega_o,\omega_i)L_i(p,\omega_i)\,|\cos\theta_i|\,d\omega_i, \tag{1}$$

where Ω is the hemisphere around the point p, ω_o is the direction of the outgoing radiance, f is the *bidirectional reflectance distribution function* (BRDF), L_i is the incoming radiance from a direction ω_i and θ_i is the angle between incoming and outgoing light directions. To approximate this equation, we will model direct lighting by light sources and lighting caused by environment effects separately:

$$L_o(p,\omega_o) \approx L_o^{\text{env}}(p,\omega_o) + \sum_{i=1}^{n} L_o^{d_i}(p,\omega_o), \tag{2}$$

where $L_o^{\text{env}}(p,\omega_o)$ is the radiance emitted from p due to lighting from a distant environment and $L_o^{d_i}(p,\omega_o)$ is the radiance emitted from p due to emission from the area light source with index i (Figure 3). Even though Equation 2 is an approximation, $L_o^{\text{env}}(p,\omega_o)$ and $L_o^{d_i}(p,\omega_o)$ are still integrals for which accurate evaluation is not feasible. Therefore, we discuss approximations of these terms using progressive rendering in the following subsections. We model the reflection of surfaces using two separate BRDFs. For the diffuse case, we assume lambertian reflection $f(\omega_o,\omega_k) = \frac{c}{\pi}$, where c is the diffuse color of the material. For specular reflections, we use the Cook-Torrance model with the GGX distribution term. The result of applying these BRDFs in a sample scene can be seen in Figure 4.

3.1 Rendering Environment Lighting Effects

In order to render environment reflections fast, we pre-filter the environment textures both the diffuse and specular BRDF. Since lambertian reflection is not view dependent, it can be accurately pre-calculated ahead of rendering time by using Monte Carlo integration. At rendering time, the pre-filtered environment map is then

[1]https://cgcostume.github.io/ermt/
[2]https://threejs.org/examples/webgl_shading_physical.html and https://threejs.org/examples/#webgl_lights_rectarealight
[3]https://connect.unity.com/p/virtualightjs-webgl-pbr-engine

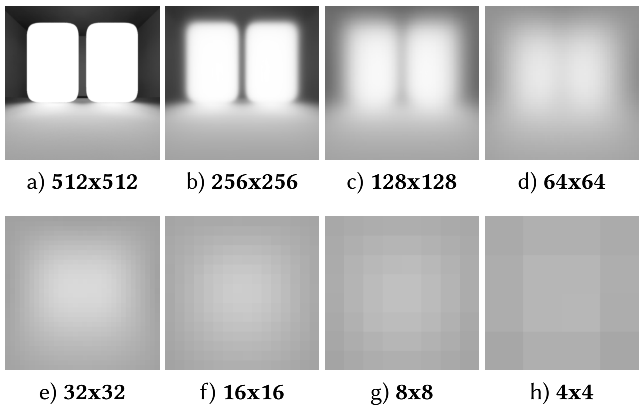


Figure 5: Mipmap levels of a face of a pre-filtered specular environment map using the GGX distribution. The images were created with our web-based tool, taking an equirectangular map as input and outputing 6 face of a cube map.

sampled using the normal vector to get an approximation of the diffuse environment lighting. For specular reflections, we use the GGX BRDF, which is view dependent, making accurate pre-filtering impossible. We use Karis' *split sum approximation* [Karis and Games 2013] to still get physically plausible results using pre-filtering. Karis approximates Monte Carlo integration in the following way:

$$L_o^{\text{env}}(p,\omega_o) \approx \frac{1}{n}\sum_{k=1}^{n} \frac{f(\omega_o,\omega_k)L_i^{\text{env}}(p,\omega_k)\,|\cos\theta_k|}{q(\omega_o,\omega_k)} \tag{3}$$

$$\approx \left(\frac{1}{n}\sum_{k=1}^{n} L_i^{\text{env}}(p,\omega_k)\right)\left(\frac{1}{n}\sum_{k=1}^{n}\frac{f(\omega_o,\omega_k)\,|\cos\theta_k|}{q(\omega_o,\omega_k)}\right). \tag{4}$$

Since the left sum is now independent of ω_o, i.e., it is not view-dependent, it can easily be pre-calculated. To enable an efficient lookup at render time, multiple resolutions are pre-calculated based different roughness values and stored in the mipmap chain of the environment texture (Figure 5). The second sum represents all other factors as if they were integrated against a fully white environment. It dependents on a materials roughness and $|\cos\theta_k|$. Therefore we pre-calculate it as a 2D texture lookup of these two values. At render time the pre-filtered environment map is sampled using the reflection vector. The appropriate mipmap level is calculated based on the material's roughness. The other sum is sampled from the lookup texture using roughness and $|\cos\theta_k|$ as texture coordinates.

3.2 Diffuse and Specular Occlusion

Using pre-filtered lookups from environment textures allows us to approximate environment lighting. However, this does not account for occlusion by nearby objects within the scene. Calculating physically correct occlusion is hard to achieve in real-time rendering and commonly approximated using *screen-space ambient occlusion* which darkens ambient lighting based on blockers in screen-space. We adapt this technique for use in a progressive rendering and, additionally, apply a similar technique to determine specular occlusion as well. We assume that depth and normal buffers of the rendered scene are available while rendering the scene. It is often sufficient to generate them only once with each camera change.

Let d_i and s_i be sets of diffuse and specular samples that are calculated during frame number i. For each sample, a tangent space vector is generated using importance sampling using the corresponding BRDF of the sample. For diffuse samples, a cosine weighted sample within the hemisphere is generated and transformed to a view space vector v. For specular samples, a half-vector is generated according to the material properties of the rendered surface and the reflection of the view vector at this half-vector is calculated to get the view space vector v. v determines the direction in which the occlusion is evaluated for this sample. It is scaled by the occlusion distance d_{occ}. Using this scaled vector, we calculate the sampling point $p = s + v$, where s is the view-space position of the shaded point. Let d_p be

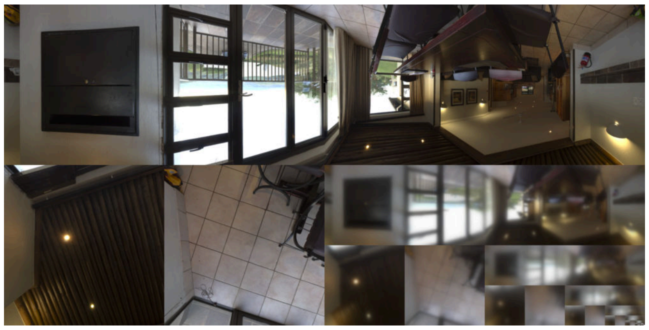


Figure 6: Efficient storage of all cube map faces of all mipmap levels within a single texture. This reduces the number of requests significantly while slightly increasing the complexity of loading individual faces. Input image—an equirectangular foto—taken from hdrihaven.com.

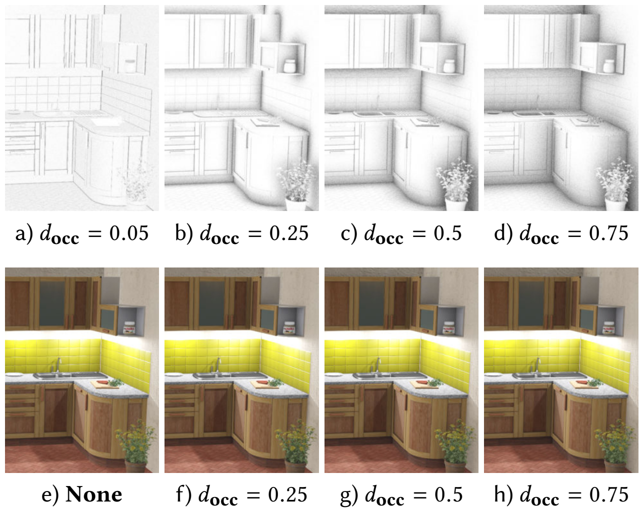
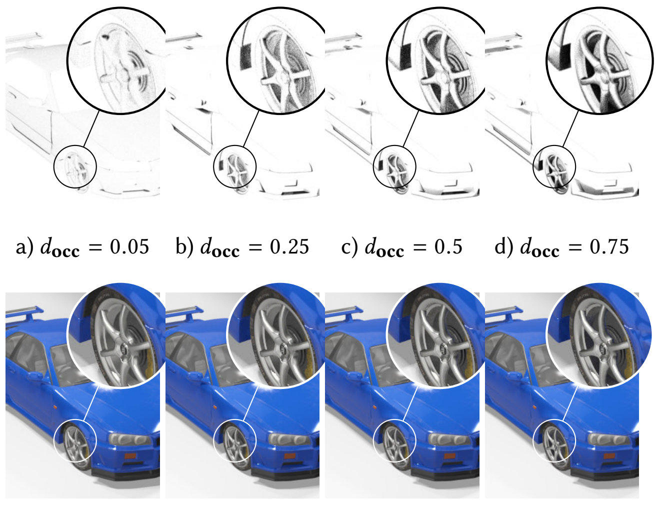

a) $d_{\text{occ}} = 0.05$
b) $d_{\text{occ}} = 0.25$
c) $d_{\text{occ}} = 0.5$
d) $d_{\text{occ}} = 0.75$
e) None
f) $d_{\text{occ}} = 0.25$
g) $d_{\text{occ}} = 0.5$
h) $d_{\text{occ}} = 0.75$

Figure 7: Rendering result of different occlusion distances d_{occ}. Top row: areas that are affected by diffuse occlusion are displayed dark, bottom row: rendering result. Note that the occlusion is less visible in the final rendering because it only affects environment lighting and not direct lighting.



a) $d_{\mathrm{occ}} = 0.05$
b) $d_{\mathrm{occ}} = 0.25$
c) $d_{\mathrm{occ}} = 0.5$
d) $d_{\mathrm{occ}} = 0.75$
e) **None**
f) $d_{\mathrm{occ}} = 0.25$
g) $d_{\mathrm{occ}} = 0.5$
h) $d_{\mathrm{occ}} = 0.75$

Figure 8: Rendering result of different occlusion distances d_{occ}. Notice that only metallic surfaces are affected by specular occlusion. Top row: areas that are affected by specular occlusion are displayed dark, bottom row: rendering result.

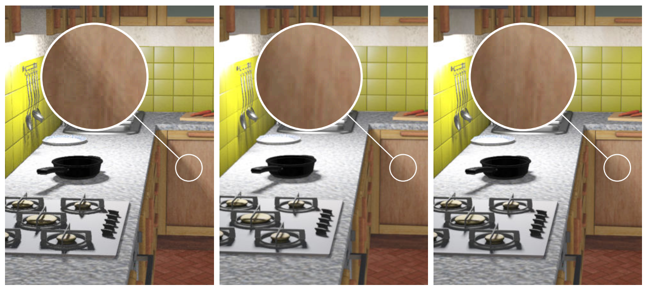


Figure 9: Kitchen scene lit by one area light using different sample counts. With 4 samples, the penumbra has an incorrect size. With 16 to 24 samples it is more accurate, but banding artifacts might remain. At 64 samples banding artifacts are gone and the penumbra has the physically correct size.

the view-space depth of p. p is transformed to normalized device coordinates and used to sample the screen-space depth d_s from the scene depth texture. Now, if $d_p > d_s$ the evaluated sample is considered to be occluding. Therefore, environment lighting from the sampled BRDF will not be added to the total scene lighting, which darkens the occluded area. The effect of diffuse and specular occlusion can be seen in Figures 7 and 8.

3.3 Progressive Area lighting

To calculate the radiance produced by an area light we have

$$L_o^{d_i}(p,\omega_o) = \int_{A_i} f(\omega_o,\omega_i) L_i^{d_i}(p,\omega_i)\,|\cos\theta_i|\,\frac{\cos\theta_o}{r^2}\,dA_i, \tag{5}$$

where we assume the outgoing radiance from a light source $L_i^{d_i}(p,\omega_i)$ to be constant, i.e., we model lights that have constant intensity across the whole area. In the context of a progressive rendering system, we can use a few samples per light source, but we are still quite limited in resources. Therefore, to calculate the intensity of specular reflections of area light sources we use Karis' [Karis and Games 2013] approximation of the *most representative point*:

$$L_o^{d_i}(p,\omega_o) \approx f(\omega_o,\omega_r) E_A(p)\,|\cos\theta_i|\,, \tag{6}$$

where ω_r is the direction pointing to the most representative point and $E_A(p)$ is the irradiance arriving at point p from area light source A. The approximation for this most representative point is the point on the area light source, which is closest to the reflection of the view vector about the normal vector. We can now approximate $E_A(p)$ by assuming A is a point light source located at the most representative point:

$$E_A(p) = \int_{\Omega} L_i^{d_i}(p,\omega_i)\,|\cos\theta_i|\,d\omega_i \tag{7}$$

$$\approx \frac{\phi_A}{4\pi d^2}, \tag{8}$$

where ϕ_A is the total power of the area light source. Combining these equations we get

$$L_o^{d_i}(p,\omega_o) \approx f(\omega_o,\omega_r)\frac{\phi_A}{4\pi d^2}\,|\cos\theta_i|\,. \tag{9}$$

To render penumbras using progressive rendering, [Limberger et al. 2016] have shown that repeated sampling of a light source over multiple frames yields believable results. We use this approach in order to approximate physically correct shadowing. We take n samples of equation 9:

$$L_o^{d_i}(p,\omega_o) \approx \frac{1}{n}\sum_{k=1}^{n} V(p,\omega_k) f(\omega_o,\omega_r)\frac{\phi_A}{4\pi d^2}\,|\cos\theta_i|\,, \tag{10}$$

where $V(p,\omega_k)$ is a visibility function using a sample ω_k on the area light source. To approximate this visibility function, we use shadow mapping, where the viewpoint of the light is changed for every sample according to the sample ω_k. Therefore, within the penumbra of the light, some samples are evaluated as occluded by the shadow mapping algorithm, while others are not. Thus, when the result is aggregated the penumbra is correctly partially shadowed. The visual impact of the number of samples is displayed in Figure 9. In order to reduce jittering across subsequent intermediate frames, it is useful to sort the samples from nearest to furthest from the center of the light source. Then, during progressive rendering first the samples with low discrepancy will be rendered and slowly more outwards samples are evaluated. Thus, the penumbras will slowly appear instead of shadows rapidly changing from frame to frame. In order to render multiple lights, let there be m number of intermediate frames and n_i number of samples for the light with index i. The samples for this light are weighted with $w_i = \frac{m}{n_i}$ in order to be correctly represented in the final rendering.

3.4 Workload Distribution

We will now discuss a strategy to distribute respective lighting samples across intermediate frames. The sampling strategy should meet the following goals:

(1) The result is physically accurate.
(2) The first frame can be rendered fast and represents the scene sufficiently accurate in order to enable navigation.
(3) The progressive rendering is temporally coherent, i.e., there are no major temporal artifacts.
(4) The sample count is adjustable and allows to favor quality or speed to account for different device capabilities.

We propose handling the first frame differently from subsequent ones by only applying environment lighting during this frame. This way, environment lighting is used as a mean to illuminate all parts of the scene at a similar level. While lighting from direct light sources is missing, the material properties are still visible when just lit by the environment and users can track their position within the scene. Since the environment lighting is cheap to calculate, especially when skipping occlusion evaluation, this speeds up calculation of the first frame. Furthermore, calculation of shadow maps and depth and normal buffers can be skipped with this approach, which further speeds up calculation.

From the set of all samples s, we want to generate subsets s_i which contain all samples that will be calculated during intermediate frame i. Ideally, these subsets should be similar in regard to how much processing power is required to calculate the contained samples. We introduce the concept of a *sample distributor*, which generates the sets s_i and issues these on requests to the renderer. With this concept, arbitrary ways to distribute samples over the course of a frame can be implemented. Let s_d be a set of all direct lighting samples and s_e a set of all environment samples calculated during a frame. We distribute s_e uniformly over the whole frame to make sure that there is an even, balanced amount of lighting across all intermediate frames. While the samples s_d should be distributed uniformly as well, we only start to evaluate them after a brief delay. This ensures that direct lighting samples, which can differ a lot from one another, do not cause flickering while the user navigates. We propose the following distribution of samples:

Algorithm 1 Distribution of samples over the course of a frame.

```
function DistributeSamples($samples, startIndex, endIndex$)
    $numFrames \leftarrow endIndex - startIndex$
    $framesPerSamples \leftarrow \frac{numFrames}{length(samples)}$
    $currentIndex \leftarrow startIndex$
    for all sample in samples do
        $i \leftarrow \lfloor currentIndex \rfloor$
        $s_i \leftarrow s_i \cap \{sample\}$
        $currentIndex \leftarrow currentIndex + framesPerSamples$
    end for
end function
```

Thus, a set of samples can be distributed within a range of intermediate frames, given the range's start and end index. With m being the number of intermediate frames, we can calculate DistributeSamples($s_e, 1, m$) and DistributeSamples($s_d, m \cdot f_d, m$), where f_d is the fraction of intermediate frames after that direct light samples start to be evaluated.

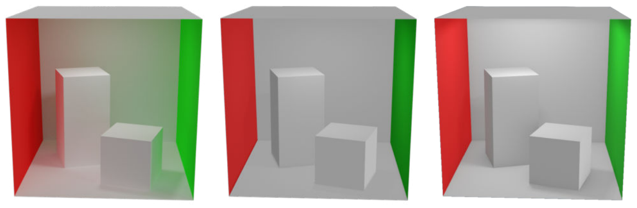


Figure 10: Rendering result for the Cornell box test scene.

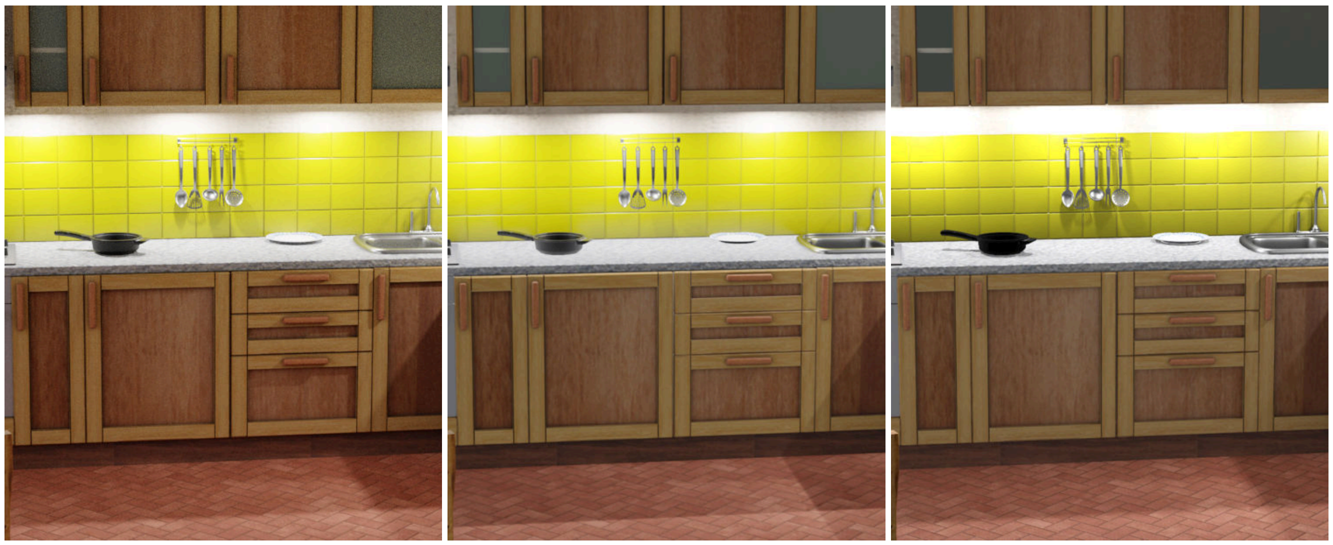
4 IMPLEMENTATION

We implemented the techniques and sampling strategies described in this work in order to verify and evaluate them. The implementation is built on top of the library *webgl-operate*[4] and written in *TypeScript*[5]. webgl-operate exposes interfaces to develop robust industry applications using WebGL. It does not enforce or favor any specific rendering approach and, furthermore, uses a generalized rendering loop that allows for continuous and progressive rendering control equally. Therefore, it is a good fit to evaluate our techniques on various devices: WebGL is available on desktops, tablets, smartphones, and other devices and support for it is growing[6]. Another benefit of webgl-operate is that tasks such as context creation, creation of WebGL objects, or fetching textures from remote URLs are encapsulated inside the library. Additionally, common rendering concepts such as *shadow mapping* can be easily used and extended due to the components already available.

We target WebGL 2.0 as it is supported on a wide range of devices and it ensures that rendering to floating point buffers is supported. This is necessary, as weighting samples in intermediate frames in combination with supporting high dynamic range rendering can lead to arbitrarily high values stored in buffers. Therefore, all rendering steps prior to applying tone mapping [Salih et al. 2012] in a post-processing pass are rendered to `gl.FLOAT` buffers. Only the result of this post-processing step is stored in a `gl.RGBA8` buffer, which is copied to the back buffer continuously for presentation to the user.

The rendering workflow is implemented in the following way. The first frame is rendered as a special case as described in Subsection 3.4, where only environment lighting is calculated. In each subsequent frame, environment and direct lighting samples are drawn from the sample distributor. These are evaluated during a forward rendering of the scene geometry. During the second frame, a normal and depth buffer is rendered, which is used for calculating occlusion effects. For each direct lighting sample, a shadow map is rendered prior to the main rendering pass to calculated shadowing for that sample. For each environment lighting sample, the diffuse or specular pre-filtered environment map is sampled and occlusion for this sample is calculated as described in Subsection 3.2.

[4]https://webgl-operate.org/
[5]https://www.typescriptlang.org/
[6]https://caniuse.com/#feat=webgl2



a) **Blender Cycles (1024 samples)** b) **Blender Eevee (64)** c) **Our renderer (256 samples)**

Figure 11: Rendering result for the kitchen test scene.

5 EVALUATION

To validate our techniques, we compare rendering results with the results of two renderers included in the 3D modelling toolkit Blender. We use the path tracer *Cycles* as a reference renderer to produce photo-realistic comparison results and the rasterizer *Eevee* as a representative modern rasterizer. We considered comparison to the progressive renderer of SketchFab[7] and the physically-based WebGL renderers by Unity[8] and the Unreal Engine[9]. However, their support for lighting is limited and especially area lights are missing. We chose 3 scenes from Sketchfab for the evaluation: the *Cornell box* by Paul, an *Italian kitchen* by F. Coldesina, and a *Nissan Skyline* by G. Rodriguez.

5.1 Evaluation of Visual Rendering Quality

The renderings of the Cornell box can be seen in Figure 10. The path-traced reference show a high amount of diffuse indirect lighting, especially at the back wall and the sides of the boxes. The two rasterizers lack this indirect lighting. The top light causes penumbras (soft shadows) from the two boxes which are not present in the Eevee rendering. In comparison, our renderer captures the shape of the shadows and matches the size of their penumbras closely to the path-traced reference. The intensity of the shadow is a bit lower compared to the reference, but more visible than in the Eevee rendering. Our progressive technique cannot perfectly capture diffuse light bounces, but handles light intensity and shadowing appropriately w.r.t. the reference in general.

The rendering results for the kitchen scene are shown in Figure 11. In the Cycles reference we can observe mostly direct lighting by six area lights above the kitchen counter. Due to multiple lights being present, we have multiple soft shadows and widening penumbras across the wooden drawers. The Eevee rendering does not correctly capture shadowing in all parts of the scene. Most notably, the shadowing of the kitchen utensils and the pan are not appropriately covered. Also the size of the penumbra on the floor is not correctly captured by the Eevee rendering. Again, our renderer matches the size and intensity of the shadows accurately and yields penumbras that are indistinguishable from the reference (without the noise).

The renderings for the Skyline (car) model are shown in Figure 12. We can see a high amount of reflection on the coating, the glass, and the tires. Because there are four lights placed on all corners of the car, shadowing is only present directly below the vehicle. Again, the Eevee rendering does not display the correct amount of shadowing. While the shadows under the car have approximately the correct shape, the intensity of the shadow is much lower than in the reference. Our renderer captures the shape and intensity of the shadows seen in the reference better than Eevee. Also, the occlusion details, e.g., at the door handle, are captured better than by the Eevee rendering. However, the bright reflections of the ground at the side are missing (see future work).

5.2 Evaluation of Rendering Performance

In order to measure the rendering speed, we perform tests on three different devices:

- A desktop computer with an Intel Core i5-3570K CPU and an AMD Radeon HD 7950 GPU.
- A 2013 MacBook Air with an Intel Core i5-4250U CPU and an Intel HD Graphics 5000 GPU.
- A Motorola Moto X4 smartphone with a Qualcomm Snapdragon 630 CPU and integrated Adreno 508 GPU.

Performance Comparison between Renderers. We compare the rendering time off all three renderers for both 1920x1080 (1080p) resolution and 1280x720 (720p) resolution. For our renderer we use a frame count of 128 for this comparison, while we use a sample

[7]https://sketchfab.com/
[8]https://unity.com/
[9]https://www.unrealengine.com/

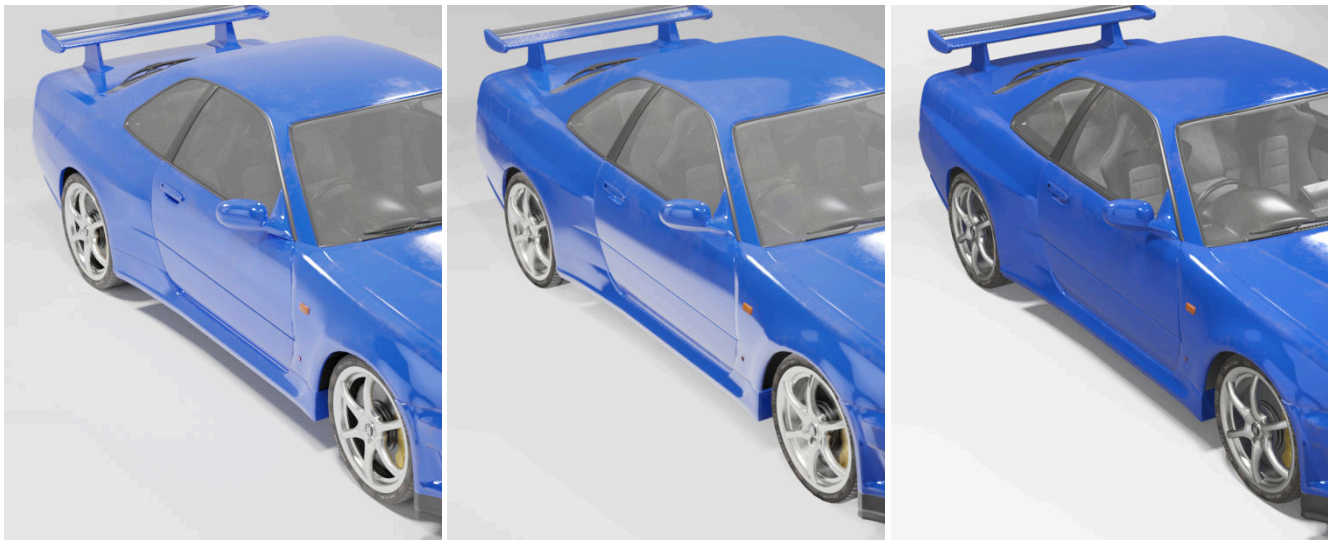

a) **Blender Cycles (1024)**
b) **Blender Eevee (64)**
c) **Our renderer (256)**

Figure 12: Rendering result for the Skyline (car) test scene.

Table 1: Rendering time in seconds of the different renderers on a desktop computer (frame count of 128 for ours).

	Cornell Box		Skyline		Kitchen	
	720p	1080p	720p	1080p	720p	1080p
Cycles	11:27.63	—	24:37.75	—	23:25.81	—
Eevee	0.53	0.99	2.38	3.37	1.77	2.55
Ours	0.19	0.41	0.65	1.39	0.82	1.68

count of 64 for Eevee. For Cycles we use a sample count of 1024 to reduce noise. The results are shown in Table 1. We can see that path tracing using Cycles takes an order of magnitude more time in comparison to the rasterizers. Despite being progressive, our renderer is faster than Eevee across all scenes and resolutions. Even with presenting intermediate results, our final rendering is available faster than using Eevee with 64 samples.

Performance Comparison on different Hardware. We run our renderer on all three devices using a frame count of 128. The results are shown in Table 2. We can see that rendering in 720p on laptop and desktop reduces rendering time significantly and even cuts the time in half in some cases. While rendering on the desktop produces results very quickly, on the laptop it takes some seconds to generate the final rendering results, especially on 1080p resolution. For a progressive renderer, this time is still acceptable because intermediate results are already shown during calculation of the image. For the smartphone, rendering more complex scenes at 720p resolution and 128 frames is not yet feasible on mid-range smartphones at the moment (about 30 seconds).

Table 2: Rendering time in seconds of our renderer on different devices using a frame count of 128.

	Cornell Box		Skyline		Kitchen	
	720p	1080p	720p	1080p	720p	1080p
Desktop	0.19	0.41	0.65	1.39	0.82	1.68
Laptop	1.26	3.21	4.58	7.52	5.44	9.85
Smartphone	7.47	—	26.27	—	32.66	—

Evaluation of Interactivity. Testing the rendering speed of the first frame for the kitchen scene resulted in 0.98, 1.28, and 11.02 milliseconds on desktop, laptop, and smartphone respectively (720p). The rendering speed is within the single digit millisecond range for both the desktop and laptop at 720p and 1080p resolution. On the smartphone we still reach an interactive rendering speed of below 16 ms. This shows that our progressive rendering with its workload distribution manages to keep the rendering time for the first frame very low.

6 CONCLUSION AND FUTURE WORK

In this work, rendering techniques and sampling strategies for lighting 3D scenes using progressive rendering were presented. Using an implementation within a web-based renderer, we showed that lighting and shadowing is physically believable, while achieving interactivity even on mobile devices. For further work, a mechanism to automatically adapt sample counts and strategies depending on the performance of the rendering device could be explored. Using an adaptive mechanism could push increase the quality of rendering on desktop devices and favor interactivity on mobile hardware. Additionally, a progressive implementation of screen-space reflections would enable specular bounces within the scene, currently not captured by our renderer.

ACKNOWLEDGMENTS

This work is part of the "Software-DNA" project, funded by the European Regional Development Fund (EFRE) and the State of Brandenburg (ILB).

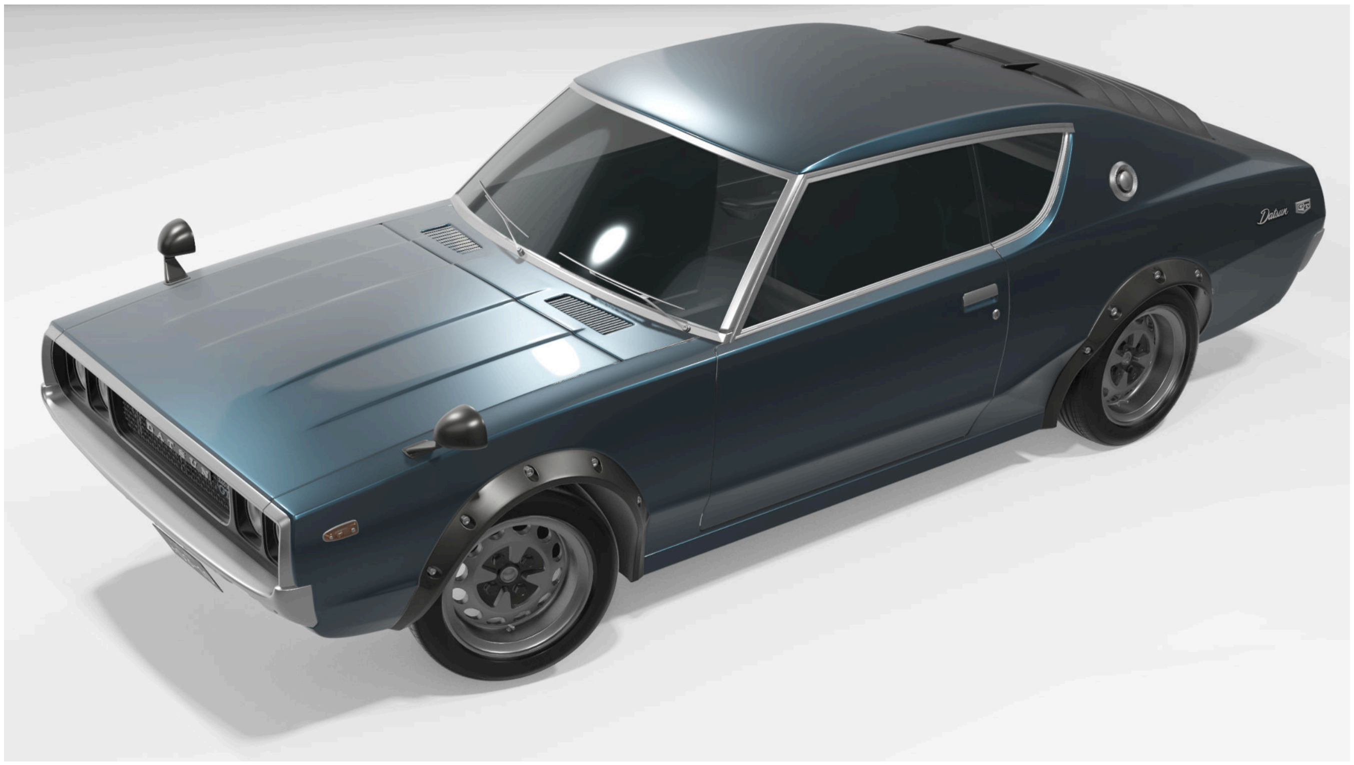


Figure 13: Our prototypical implementation available at https://p-otto.waduhek.de/demos/progressive-lighting.html. We are currently adding on screen-space reflections and depth-of-field to the renderer. 3D Model: *Datsun 240k GT* by Karol Miklas

.

REFERENCES

James F Blinn and Martin E Newell. 1976. Texture and reflection in computer generated images. *Commun. ACM* 19, 10 (1976), 542–547.

Brent Burley and Walt Disney Animation Studios. 2012. Physically Based Shading at Disney. *SIGGRAPH 2012 Courses: Practical Physically Based Shading in Film and Game Production.*

Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 98.

Mark Colbert, Simon Premoze, and Guillaume François. 2010. Importance sampling for production rendering. *SIGGRAPH 2010 Course Notes* (2010), 19.

Robert L Cook and Kenneth E Torrance. 1982. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)* 1, 1 (1982), 7–24.

Henry Fuchs, Jack Goldfeather, Jeff P Hultquist, Susan Spach, John D Austin, Frederick P Brooks Jr, John G Eyles, and John Poulton. 1985. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. *ACM SIGGRAPH Computer Graphics* 19, 3 (1985), 111–120.

Ned Greene. 1986. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications* 6, 11 (1986), 21–29.

Paul Haeberli and Kurt Akeley. 1990. The accumulation buffer: hardware support for high-quality rendering. *ACM SIGGRAPH computer graphics* 24, 4 (1990), 309–318.

Dean Jackson and Jeff Gilbert. 2015. WebGL 2.0 Specification. *Khronos Group* (2015).

Jorge Jiménez, Jose I Echevarria, Tiago Sousa, and Diego Gutierrez. 2012. SMAA: enhanced subpixel morphological antialiasing. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 355–364.

Jorge Jiménez, X Wu, Angelo Pesce, and Adrian Jarabo. 2016. Practical real-time strategies for accurate indirect occlusion. *SIGGRAPH 2016 Courses: Physically Based Shading in Theory and Practice* (2016).

B Karis. 2014. High Quality Temporal Anti-Aliasing. *Advances in Real-Time Rendering for Games, SIGGRAPH Courses* (2014).

Brian Karis and Epic Games. 2013. Real Shading in Unreal Engine 4. *Proceedings of Physically Based Shading in Theory and Practice, SIGGRAPH 2013 Course Notes* (2013).

Sebastien Lagarde and Charles De Rousiers. 2014. Moving Frostbite to PBR. *Proceedings of Physically Based Shading in Theory and Practice, SIGGRAPH 2014 Course Notes* (2014).

Daniel Limberger, Marcel Pursche, Jan Klimke, and Jürgen Döllner. 2017. Progressive high-quality rendering for interactive information cartography using WebGL. In *Proc. ACM International Conference on 3D Web Technology*. 1–4.

Daniel Limberger, Karsten Tausche, Johannes Linke, and Jürgen Döllner. 2016. Progressive rendering using multi-frame sampling. *GPU Pro 7: Advanced Rendering Techniques* (2016), 155.

Martin Mittring. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*. ACM, 97–121.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann.

Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. 2009. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 75–82.

Fabrice Robinet and P Cozzi. 2013. glTF — The Runtime Asset Format for WebGL, OpenGL ES, and OpenGL.

Yasir Salih, Aamir S Malik, Naufal Saad, et al. 2012. Tone mapping of HDR images: A review. In *2012 4th International Conference on Intelligent and Advanced Systems (ICIAS2012)*, Vol. 1. IEEE, 368–373.

Thorsten Scheuermann and John Isidoro. 2006. Cubemap Filtering with CubeMapGen. In *Game Developer Conference 2006.*

Timo Sturm, Miguel Sousa, Maik Thöner, and Max Limper. 2016. A unified GLTF/X3D extension to bring physically-based rendering to the web. In *Proceedings of the 21st International Conference on Web3D Technology*. 117–125.

Marc Sunet and Pere-Pau Vazquez. 2016. Optimized screen-space ambient occlusion in mobile devices. In *Proc. ACM International Conference on Web3D Technology*. 127–135.

Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. 2007. Microfacet models for refraction through rough surfaces. In *Proc. EG Conference on Rendering Techniques*. Eurographics Association, 195–206.